■ ■ ■

Introducing Java SE 6

Java SE 6, the sixth generation of Java Standard Edition since version 1.0, officially arrived on December 11, 2006. This release offers many features that will benefit Java developers for years to come. This chapter introduces you to Java SE 6 and some of its features via the following topics:

- Name change for this Java edition

- Themes of Java SE 6

- Overview of Java SE 6

- Sampling of Java SE 6 new features

- Java SE 6, update 1 and update 2

■**Tip** Meet the developers behind Java SE 6 by visiting the Planet JDK site (`http://planetjdk.org/`), which was created by Java SE Chief Engineer Mark Reinhold (see "Announcing planetjdk.org" at `http://weblogs.java.net/blog/mreinhold/archive/2005/11/announcing_plan.html`). You can learn a lot about Java SE 6 by reading the developers' blogs and articles. I present links to relevant blog and article entries throughout this book.

Name Change for This Java Edition

At different times during Java's 12-year history, Sun has introduced a new naming convention for its assorted Java editions, development kits, and runtime environments. For example, Java Development Kit (JDK) 1.2 became known as Java 2 Platform, Standard Edition 1.2 (J2SE 1.2). More recently, Sun announced that the fifth generation of its standard edition (since JDK 1.0) would be known as Java 2 Platform, Standard Edition 5.0 (J2SE 5.0), instead of the expected Java 2 Platform, Standard Edition 1.5.0 (J2SE 1.5.0).

The 5.0 is known as the external version number, and 1.5.0 is used as the internal version number.

Prior to releasing the latest generation, Sun's marketing team met with a group of its Java partners, and most agreed to simplify the Java 2 Platform's naming convention to build brand awareness. In addition to dropping the *2* from Java 2 Platform, Standard Edition, the "dot number" (the number following the period, as in 5.0) would be dropped, so that future updates to the Java platform would be noted as updates rather than dot numbers tacked onto the end of platform names. Hence, this latest Java release is known as *Java Platform, Standard Edition 6* (*Java SE 6*).

Similar to the 5.0 in J2SE 5.0 (which I refer to as Java 5 throughout this book), 6 is the external version number in the latest release. Also, 1.6.0 is the internal version number, which appears in the various places identified on Sun's Java SE 6, Platform Name and Version Numbers page (`http://java.sun.com/javase/6/webnotes/version-6.html`). This page also indicates that JDK (which now stands for Java SE Development Kit) continues to be the acronym for the development kit, and JRE continues to be the acronym for the Java Runtime Environment.

■**Note** Jon Byous discusses the new naming convention in more detail via his "Building and Strengthening the Java Brand" article (`http://java.sun.com/developer/technicalArticles/JavaOne2005/naming.html`). Also, check out Sun's "New! Java Naming Gets a Birthday Present" article (`http://www.java.com/en/about/brand/naming.jsp`).

The Themes of Java SE 6

Java SE 6 was developed under Java Specification Request (JSR) 270 (`http://jcp.org/en/jsr/detail?id=270`), which presents the themes listed in this section. The themes are also mentioned in Sun's official press release on Java SE 6, "Sun Announces Revolutionary Version of Java Technology – Java Platform Standard Edition 6" (`http://www.sun.com/smi/Press/sunflash/2006-12/sunflash.20061211.1.xml`).

Compatibility and stability: Many members of the Java community have invested heavily in Java technology. Because it is important that their investments are preserved, effort has been expended to ensure that the vast majority of programs that ran on previous versions of the Java platform continue to run on the latest platform. A few programs may need to be tinkered with to get them to run, but these should be rare. Stability is just as important as compatibility. Many bugs have been fixed, and the HotSpot virtual machines and their associated runtime environments are even more stable in this release.

Diagnosability, monitoring, and management: Because Java is widely used for mission-critical enterprise applications that must be kept running, it is important to have support for remote monitoring, management, and diagnosis. To this end, Java SE 6 improves the existing Java Management Extensions (JMX) API and infrastructure, as well as JVM Tool Interface. For example, you now have the ability to monitor applications not started with a special monitoring flag (you can look inside any running application to see what is happening under the hood).

Ease of development: Java SE 6 simplifies a developer's life by providing new annotation types, such as `@MXBean` for defining your own MBeans; a scripting framework that you can use to leverage the advantages offered by JavaScript, Ruby, and other scripting languages; redesigned Java Database Connectivity (JDBC) that benefits from automatic driver loading; and other features.

Enterprise desktop: As developers encounter the limitations of browser-based thin clients, they are once again considering rich client applications. To facilitate the migration to rich client applications, Java SE 6 provides better integration with native desktop facilities (such as the system tray, access to the default web browser and other desktop helper applications, and splash screens), the ability to print the contents of text components, the ability to sort and filter table rows, font anti-aliasing so that text is more readable on liquid crystal display (LCD) screens, and more.

XML and web services: Java SE 6 provides significant enhancements in the area of XML; XML digital signatures and Streaming API for XML (StAX) are two examples. Although Java 5 was supposed to include a web services client stack, work on this feature could not be finished in time for Java 5's release. Fortunately, Java SE 6 includes this stack—hello, Web 2.0!

Transparency: According to JSR 270, "Transparency is new and reflects Sun's ongoing effort to evolve the J2SE platform in a more open and transparent manner." This is in response to the desire of many developers to participate more fully in the development of the next generation of Java. Because of the positive reception to Sun's "experiment in openness"—making Java 5 (Tiger) snapshot releases available to the public, which allowed developers to collaborate with Sun on fixing problems—Sun enhanced this experiment for Java SE 6. This transparency has fully evolved into Sun open-sourcing the JDK. Developers now have more influence on the features to be made available in the next generation of Java.

■**Note** For more information about Java SE 6 transparency and open-sourcing, see Java SE Chief Engineer Mark Reinhold's "Mustang Snapshots: Another experiment in openness" blog entry (`http://weblogs.java.net/blog/mreinhold/archive/2004/11/index.html`) and the OpenJDK Community page (`http://community.java.net/openjdk/`).

Not every Java SE 6 feature is associated with a theme. For example, the class file specification update does not belong to any of the aforementioned themes. Also, not every theme corresponds to a set of features. For example, transparency reflects Sun's desire to be more open in how it interacts with the Java community while developing a platform specification and the associated reference implementation. Also, compatibility constrains how the platform evolves, because evolution is limited by the need to remain compatible with previous releases to support the existing base of Java software.

Overview of Java SE 6

Java SE 6 (which was formerly known by the code name Mustang during development) enhances the Java platform via improvements to the platform's performance and stability, by fixing assorted bugs, and even improvements to make graphical user interfaces (GUIs) look better (anti-aliasing LCD text is an example). Java SE 6 also enhances the Java platform by introducing a rich set of completely new features, some of which I've already mentioned. Many of these new features were developed by the various component JSRs of JSR 270, which serves as the "umbrella" JSR for Java SE 6:

- JSR 105: XML Digital Signature APIs (`http://jcp.org/en/jsr/detail?id=105`)

- JSR 199: Java Compiler API (`http://jcp.org/en/jsr/detail?id=199`)

- JSR 202: Java Class File Specification Update (`http://jcp.org/en/jsr/detail?id=202`)

- JSR 221: JDBC 4.0 API Specification (`http://jcp.org/en/jsr/detail?id=221`)

- JSR 222: Java Architecture for XML Binding (JAXB) 2.0 (`http://jcp.org/en/jsr/detail?id=222`)

- JSR 223: Scripting for the Java Platform (`http://jcp.org/en/jsr/detail?id=223`)

- JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0 (`http://jcp.org/en/jsr/detail?id=224`)

- JSR 268: Java Smart Card I/O API (`http://jcp.org/en/jsr/detail?id=268`)

- JSR 269: Pluggable Annotation Processing API (`http://jcp.org/en/jsr/detail?id=269`)

The one JSR specified in JSR 270's list of component JSRs that was not included in Java SE 6 is JSR 260: Javadoc Tag Technology Update (`http://jcp.org/en/jsr/detail?id=260`). Additional JSRs not specified in JSR 270's list, but that did make it into Java SE 6, are as follows:

- JSR 173: Streaming API for XML (`http://jcp.org/en/jsr/detail?id=173`)

- JSR 181: Web Services Metadata for the Java Platform (`http://jcp.org/en/jsr/detail?id=181`)

- JSR 250: Common Annotations for the Java Platform (`http://jcp.org/en/jsr/detail?id=250`)

Although these JSRs provide insight into what has been included in Java SE 6, "What's New in Java SE 6" (`http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/beta2.html`) offers a more complete picture. This article presents Danny Coward's "Top 10 Things You Need to Know" list of new Java SE 6 features (Danny Coward is the platform lead for Java SE), and Mark Reinhold's table of approved features. Of the table's listed features, internationalized resource identifiers (IRIs), the ability to highlight a `javax.swing.JTable`'s rows, and reflective access to parameter names did not make it into Java SE 6. IRIs, explained in RFC 3987: Internationalized Resource Identifiers (IRIs) (`http://www.ietf.org/rfc/rfc3987.txt`) were removed from the final release of Java SE 6 as part of `java.net.URI` being rolled back to the Java 5 version; see Bug 6394131 "Rollback URI class to Tiger version" in Sun's Bug Database").

■**Note** The JDK 6 documentation's main page (`http://java.sun.com/javase/6/docs/`) presents a New Features and Enhancements link to the Features and Enhancements page (`http://java.sun.com/javase/6/webnotes/features.html`), which has more information about what is new and improved in Java SE 6.

Sampling of Java SE 6 New Features

As you will have noticed from the various feature references in the previous two sections, Java SE 6 has a lot to offer. This book explores most of Java SE 6's new and improved features, ranging from enhancements to the core libraries to a variety of performance enhancements. Before moving on, let's sample some of the features that set Java SE 6 apart from its predecessors.

A Trio of New Action Keys and a Method to Hide/Show Action Text

The `javax.swing.Action` interface extends the `java.awt.event.ActionListener` interface to bundle, in the same class, several component properties such as `toolTipText` and `icon` with common code. An instance of this class can be attached to multiple components (an Open menu item on a File menu and an Open button on a toolbar, for example), which then can be enabled/disabled from one place. Furthermore, selecting either component executes the common code. Java SE 6 lets you manipulate two new properties and a variation of `icon` via these new keys:

- `DISPLAYED_MNEMONIC_INDEX_KEY`: Identifies the index in the `text` property (accessed via the `NAME` key) where a mnemonic decoration should be rendered. This key corresponds to the new `displayedMnemonicIndex` property; the key's associated value is an `Integer` instance.

- `LARGE_ICON_KEY`: Identifies the `javax.swing.Icon` that appears on various kinds of Swing buttons, such as an instance of `javax.swing.JButton`. The `javax.swing.JMenuItem` subclasses, such as `javax.swing.JCheckBoxMenuItem`, use the `Icon` associated with the `SMALL_ICON` key. Unlike `LARGE_ICON_KEY`, there is no `SMALL_ICON_KEY` constant with a `_KEY` suffix.

- `SELECTED_KEY`: Initializes the selection state of a toggling component, such as an instance of `javax.swing.JCheckBox`, from an action and reflects this change in the component. This key corresponds to the new `selected` property; the key's associated value is a Boolean instance.

Java SE 6 also adds new action-related `public void setHideActionText(boolean hideActionText)` and `public boolean getHideActionText()` methods to the `javax.swing.AbstractButton` class. The former method sets the value of the `hideActionText` property, which determines whether (`true` passed to `hideActionText`) or not (`false` passed to `hideActionText`) a button displays an action's text; by default, a toolbar button does not display this text. The latter method returns this property's current setting. Listing 1-1 presents a notepad application that demonstrates these new action keys and methods.

Listing 1-1. *Notepad.java*

```java
// Notepad.java

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
```

```java
import javax.swing.border.*;

public class Notepad extends JFrame
{
   private JTextArea document = new JTextArea (10, 40);

   public Notepad ()
   {
      super ("Notepad 1.0");
      setDefaultCloseOperation (EXIT_ON_CLOSE);

      JMenuBar menuBar = new JMenuBar ();

      JToolBar toolBar = new JToolBar ();

      JMenu menu = new JMenu ("File");
      menu.setMnemonic (KeyEvent.VK_F);

      Action newAction = new NewAction (document);
      menu.add (new JMenuItem (newAction));
      toolBar.add (newAction);

      // Java SE 6 introduces a setHideActionText() method to determine
      // whether or not a button displays text originating from an action. To
      // demonstrate this method, the code below makes it possible for a
      // toolbar button to display the action's text -- a toolbar button does
      // not display this text in its default state.

      JButton button = (JButton) toolBar.getComponentAtIndex (0);
      button.setHideActionText (false);

      menuBar.add (menu);

      menu = new JMenu ("View");
      menu.setMnemonic (KeyEvent.VK_V);

      Action statAction = new StatAction (this);
      menu.add (new JCheckBoxMenuItem (statAction));

      menuBar.add (menu);

      setJMenuBar (menuBar);
```

```java
      getContentPane ().add (toolBar, BorderLayout.NORTH);
      getContentPane ().add (document, BorderLayout.CENTER);

      pack ();
      setVisible (true);
   }

   public static void main (String [] args)
   {
      Runnable r = new Runnable ()
                {
                    public void run ()
                    {
                        new Notepad ();
                    }
                };
      EventQueue.invokeLater (r);
   }
}

class NewAction extends AbstractAction
{
   JTextArea document;

   NewAction (JTextArea document)
   {
      this.document = document;

      putValue (NAME, "New");
      putValue (MNEMONIC_KEY, KeyEvent.VK_N);
      putValue (SMALL_ICON, new ImageIcon ("newicon_16x16.gif"));

      // Before Java SE 6, an action's SMALL_ICON key was used to assign the
      // same icon to a button and a menu item. Java SE 6 now makes it
      // possible to assign different icons to these components. If an icon
      // is added via LARGE_ICON_KEY, this icon appears on buttons, whereas
      // an icon added via SMALL_ICON appears on menu items. However, if there
      // is no LARGE_ICON_KEY-based icon, the SMALL_ICON-based icon is
      // assigned to a toolbar's button (for example), in addition to a menu
      // item.
```

```
      putValue (LARGE_ICON_KEY, new ImageIcon ("newicon_32x32.gif"));
   }

   public void actionPerformed (ActionEvent e)
   {
      document.setText ("");
   }
}

class StatAction extends AbstractAction
{
   private JFrame frame;
   private JLabel labelStatus = new JLabel ("Notepad 1.0");

   StatAction (JFrame frame)
   {
      this.frame = frame;

      putValue (NAME, "Status Bar");
      putValue (MNEMONIC_KEY, KeyEvent.VK_A);

      // By default, a mnemonic decoration is presented under the leftmost
      // character in a string having multiple occurrences of this character.
      // For example, the previous putValue (MNEMONIC_KEY, KeyEvent.VK_A);
      // results in the "a" in "Status" being decorated. If you prefer to
      // decorate a different occurrence of a letter (such as the "a" in
      // "Bar"), you can now do this thanks to Java SE 6's
      // displayedMnemonicIndex property and DISPLAYED_MNEMONIC_INDEX_KEY. In
      // the code below, the zero-based index (8) of the "a" appearing in
      // "Bar" is chosen as the occurrence of "a" to receive the decoration.

      putValue (DISPLAYED_MNEMONIC_INDEX_KEY, 8);

      // Java SE 6 now makes it possible to choose the initial selection state
      // of a toggling component. In this application, the component is a
      // JCheckBoxMenuItem that is responsible for determining whether or not
      // to display a status bar. Initially, the status bar will not be shown,
      // which is why false is assigned to the selected property in the method
      // call below.
```

```
    putValue (SELECTED_KEY, false);

    labelStatus = new JLabel ("Notepad 1.0");
    labelStatus.setBorder (new EtchedBorder ());
  }

  public void actionPerformed (ActionEvent e)
  {
     // Because a component updates the selected property, it is easy to find
     // out the current selection setting, and then use this setting to
     // either add or remove the status bar.

     Boolean selection = (Boolean) getValue (SELECTED_KEY);
     if (selection)
         frame.getContentPane ().add (labelStatus, BorderLayout.SOUTH);
     else
         frame.getContentPane ().remove (labelStatus);

     frame.getRootPane ().revalidate ();
  }
}
```

The numerous comments in the source code explain the new action keys and the
setHideActionText() method. However, you might be curious about my deferring the
creation of a Swing application's GUI to the event-dispatching thread, via a Runnable
instance and the EventQueue.invokeLater (r); method call. I do this here (and elsewhere
in the book) because creating a Swing GUI on any thread other than the event-dispatching
thread—such as an application's main thread or the thread that invokes an applet's public
void init() method—is unsafe.

■**Note** Although you could invoke SwingUtilities.invokeLater() to ensure that an application's
Swing-based GUI is created on the event-dispatching thread, it is somewhat more efficient to invoke
EventQueue.invokeLater(), because the former method contains a single line of code that calls the
latter method. It is also somewhat more efficient to invoke EventQueue.invokeAndWait(), rather than
SwingUtilities.invokeAndWait(), to create an applet's Swing-based GUI on the event-dispatching
thread.

Creating a Swing GUI on a thread other than the event-dispatching thread is unsafe because the Swing GUI toolkit is not multithreaded. (Check out Graham Hamilton's "Multithreaded toolkits: A failed dream?" blog entry at `http://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html` to find out why Swing is not multithreaded.) As a result, creating the GUI on the main thread while the event-dispatching thread is also running potentially leads to problems that might or might not be difficult to solve.

For example, suppose you create the GUI on the main thread, and part of the GUI-creation code indirectly creates a `javax.swing.text.JTextComponent` via some subclass, such as `javax.swing.JEditorPane`. `JTextComponent` includes several methods that call `invokeLater();` the `public void insertUpdate(DocumentEvent e)` event-handling method is an example. If this method should somehow be invoked during GUI creation, its call to `invokeLater()` would result in the event-dispatching thread starting (unless that thread is already running). The application would then be in a position where the integrity of the Swing GUI toolkit is violated.

According to older versions of *The Java Tutorial*, Swing GUIs could be created on threads other than the event-dispatching thread. This advice is also detailed in Hans Muller's and Kathy Walrath's older "Threads and Swing" article (`http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`). In contrast, the latest version of *The Java Tutorial* insists on creating the GUI on the event-dispatching thread (see `http://java.sun.com/docs/books/tutorial/uiswing/concurrency/initial.html`).

■**Note** Tech writer Cay Horstmann's "The Single Thread Rule in Swing" blog entry (`http://weblogs.java.net/blog/cayhorstmann/archive/2007/06/the_single_thre.html`) provides an interesting read (especially in the comments section) of the create-Swing-GUI-on-event-dispatching-thread topic.

The notepad application in Listing 1-1 requires more work to turn it into something useful. However, it does serve the purpose of demonstrating these new action keys and methods. For example, after compiling `Notepad.java` and running this application, you'll notice the result of the `setHideActionText()` method: *New* on the toolbar icon. Also, when you open the File menu, you'll notice a different (and smaller) icon appearing beside the New menu item. Figure 1-1 shows the application's GUI with these enhancements. In the figure, I've moved the toolbar to the right so that you can easily see the two different icons. Of course, you will typically not display text on toolbar buttons that also present images.
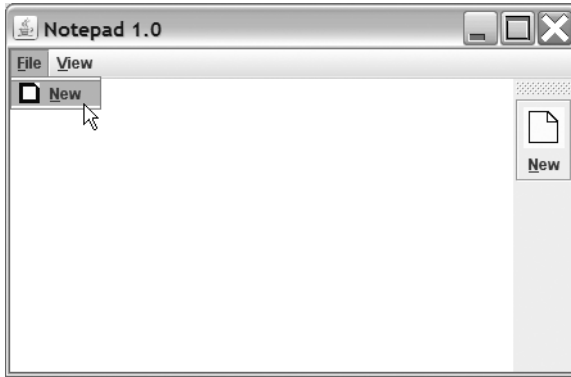
Figure 1-1. *The setHideActionText() method made it possible for New to appear with the icon on the toolbar button.*

■**Note** DISPLAYED_MNEMONIC_INDEX_KEY, LARGE_ICON_KEY, SELECTED_KEY, and the setHideActionText() method are discussed in Scott Violet's "Changes to Actions in 1.6" blog entry (http://weblogs.java.net/blog/zixle/archive/2005/11/changes_to_acti.html). Scott's blog entry also discusses the swing.actions.reconfigureOnNull system property.

Clearing a ButtonGroup's Selection

You create a form-based GUI that includes a group of radio buttons, with none of these buttons initially selected. When the user clicks the form's Reset button, you want to clear any selected radio button in this group (no radio button should be selected). According to Java 5's JDK documentation for javax.swing.ButtonGroup:

> *There is no way to turn a button programmatically to "off," in order to clear the button group. To give the appearance of "none selected," add an invisible radio button to the group and then programmatically select that button to turn off all the displayed radio buttons. For example, a normal button with the label "none" could be wired to select the invisible radio button.*

The documentation's advice results in extra code that complicates the GUI design, and probably leads to GUI logic that is difficult to follow. Although it seems that passing false to ButtonGroup's public void setSelected(ButtonModel m, boolean b) method should do the trick, the method's source code recognizes only a true value. Fortunately, Java SE 6 comes to the rescue. In response to Bug 4066394 "ButtonGroup – cannot reset the model

to the initial unselected state," Java SE 6 adds a new `public void clearSelection()` method to `ButtonGroup`. According to the JDK 6 documentation, this method "clears the selection such that none of the buttons in the `ButtonGroup` are selected."

Enhancements to Reflection

Java SE 6 enhances Java's support for reflection as follows:

- By fixing the `public String toGenericString()` and `public string toString()` methods in the `java.lang.reflect.Method` and `java.lang.reflect.Constructor` classes to correctly display modifiers

- By modifying the final parameter in Java 5's `public static Object newInstance(Class<?> componentType, int[] dimensions)` method to use variable arguments; the new method signature is `public static Object newInstance(Class<?> componentType, int... dimensions)`

- By generifying the following methods of `Class`:

    - `public Class<?>[] getClasses()`

    - `public Constructor<T> getConstructor(Class<?>... parameterTypes)`

    - `public Constructor<?>[] getConstructors()`

    - `public Class<?>[] getDeclaredClasses()`

    - `public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)`

    - `public Constructor<?>[] getDeclaredConstructors()`

    - `public Method getDeclaredMethod(String name, Class<?>... parameterTypes)`

    - `public Class<?>[] getInterfaces()`

    - `public Method getMethod(String name, Class<?>... parameterTypes)`

The problems with the `toGenericString()` and `toString()` methods in Java 5's `Method` and `Constructor` classes are documented by Bug 6261502 (reflect) "Add the functionality to screen out the 'inappropriate' modifier bits," Bug 6316717 (reflect) "Method.toGenericString prints out inappropriate modifiers," Bug 6354476 (reflect) "{Method, Constructor}.toString prints out inappropriate modifiers," and Bug 6362451 "The string returned by toString() shows the bridge methods as having the volatile modificator."

■**Note** During Java SE 6's development, consideration was given to enhancing Java's reflection capability by supporting reflective access to constructor and method parameter names. Although this feature did not make it into Java SE 6, it could make it into the next release. If you are curious about this feature, check out Andy Hedges' "Reflective Access to Parameter Names" blog entry (`http://hedges.net/archives/ 2006/04/07/reflective-access-to-parameter-names/`).

GroupLayout Layout Manager

Java SE 6 adds *GroupLayout* to its suite of layout managers. GroupLayout hierarchically groups components in order to position them within a container. It consists of the `javax.swing.GroupLayout` class (and inner classes) and the `GroupLayout.Alignment` enumeration. The `GroupLayout` class works with the new `javax.swing.LayoutStyle` class to obtain component-positioning information, as well as the `java.awt.Component` class's new `public Component.BaselineResizeBehavior getBaselineResizeBehavior()` and `public int getBaseline(int width, int height)` methods.

■**Note** According to its JDK documentation, the `BaselineResizeBehavior` enumeration enumerates "the common ways the baseline of a component can change as the size changes." For example, the baseline remains a fixed distance from the component's center as the component is resized. GroupLayout invokes the `getBaselineResizeBehavior()` method when it needs to know the specific resize behavior. When it needs to identify the baseline from the top of the component, GroupLayout invokes `getBaseline()`.

Although this layout manager is intended for use with GUI builders (such as the Matisse GUI builder in NetBeans 5.5), GroupLayout also can be used to manually code layouts. If you are interested in learning how to do this, you should check out the "How to Use GroupLayout" section (`http://java.sun.com/docs/books/tutorial/uiswing/ layout/group.html`) in *The Java Tutorial*'s "Laying Out Components Within a Container" lesson. You should also check out the `GroupLayout` class's JDK documentation.

■**Note** GroupLayout originated as an open-source project at java.net's swing-layout project site (`http://swing-layout.dev.java.net`). Because of its success with NetBeans 5.0, GroupLayout was merged into Java SE 6 (with various changes, primarily in the area of package and method names). Although NetBeans 5.0 supports only the swing-layout version, NetBeans 5.5 supports the swing-layout version for pre-Java SE 6 and the Java SE 6 version for Java SE 6.

Image I/O GIF Writer Plug-in

For years, developers have wanted the Image I/O framework to provide a plug-in for writing images in the GIF file format—see Bug 4339415 "Provide a writer plug-in for the GIF file format." However, it was not possible to provide this plug-in as long as any Unisys patents on the Lempel-Ziv-Welch data compression algorithm used in writing GIF files remained in effect. Because Unisys's final international patents (Japanese patents 2,123,602 and 2,610,084) expired on June 20, 2004 (see "Sad day . . . GIF patent dead at 20," `http://www.kuro5hin.org/story/2003/6/19/35919/4079`), it finally became possible to add this plug-in to Image I/O. Java SE 6 includes a GIF writer plug-in. Listing 1-2 presents an application that uses this plug-in to write a simple image to `image.gif`.

Listing 1-2. *SaveToGIF.java*

```
// SaveToGIF.java

import java.awt.*;
import java.awt.image.*;

import java.io.*;

import javax.imageio.*;

public class SaveToGIF
{
   final static int WIDTH = 50;
   final static int HEIGHT = 50;
   final static int NUM_ITER = 1500;

   public static void main (String [] args)
   {
      // Create a sample image consisting of randomly colored pixels in
      // randomly colored positions.

      BufferedImage bi;
      bi = new BufferedImage (WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
      Graphics g = bi.getGraphics ();
      for (int i = 0; i < NUM_ITER; i++)
      {
          int x = rnd (WIDTH);
          int y = rnd (HEIGHT);
          g.setColor (new Color (rnd (256), rnd (256), rnd (256)));
```

```
            g.drawLine (x, y, x, y);
        }
        g.dispose ();

        // Save the image to image.gif.

        try
        {
            ImageIO.write (bi, "gif", new File ("image.gif"));
        }
        catch (IOException ioe)
        {
            System.err.println ("Unable to save image to file");
        }
    }

    static int rnd (int limit)
    {
        return (int) (Math.random ()*limit);
    }
}
```

Beyond the GIF writer plug-in, Java SE 6 improves Image I/O performance. Appendix C provides the details.

Incremental Improvements to String

The String class has slightly improved in Java SE 6. New public String(byte[] bytes, int offset, int length, Charset charset) and public String(byte[] bytes, Charset charset) constructors have been added as alternatives to the equivalent public String(byte[] bytes, int offset, int length, String charsetName) and public String(byte[] bytes, Charset charset) constructors. As pointed out in Bug 5005831 "String constructors and method which take Charset rather than String as argument," these older constructors were found to be inefficient at converting from bytes to strings, which is a common operation in I/O-bound applications, especially in a server-side environment. To complement these constructors, a new public byte[] getBytes(Charset charset) method has been introduced as a more efficient alternative to the public byte[] getBytes(String charsetName) method.

Finally, a new public boolean isEmpty() method has been added, in response to Bug 6189137 "New String convenience methods isEmpty() and contains(String)." This method returns true if the String's length equals 0.

■**Note** Contrary to what appears in Sun's *Java SE 6 – In Depth Overview* PDF-based document (`https://java-champions.dev.java.net/pdfs/SE6-in-depth-overview.pdf`), `String`'s `indexOf()` and `lastIndexOf()` methods have not been enhanced to support the Boyer-Moore algorithm for faster searching. The Java SE 6 source code for these methods is the same as the Java 5 source code. For the rationale in not supporting Boyer-Moore, check out Bug 4362107 "String.indexOf(String) needlessly inefficient."

LCD Text Support

The "LCD Text" section of tech writer Robert Eckstein's "New and Updated Desktop Features in Java SE 6, Part 1" article (`http://java.sun.com/developer/technicalArticles/javase/6_desktop_features/index.html`) describes a new Java SE 6 feature for improving text resolution on LCDs. This feature is an LCD text algorithm that anti-aliases text (to smooth edges) for presentation on LCDs. The anti-aliased text looks better and is easier to read, as evidenced by the article's screenshots. (You will need an appropriate display configuration, as explained in the article, to see the improvement offered by these images.) Because the Metal, GTK, and Windows look and feels automatically support LCD text, applications that use these look and feels benefit from this feature.

■**Note** Chet Haase provides an excellent introduction to LCD text in his "LCD Text: Anti-Aliasing on the Fringe" article (`http://today.java.net/pub/a/today/2005/07/26/lcdtext.html`).

If you are developing a custom look and feel, and want it to take advantage of LCD text, you will need to acquaint yourself with the `java.awt.RenderingHints` class's `KEY_TEXT_ANTIALIASING` key constant, and its `VALUE_TEXT_ANTIALIAS_LCD_HRGB`, `VALUE_TEXT_ANTIALIAS_LCD_HBGR`, `VALUE_TEXT_ANTIALIAS_LCD_VRGB`, `VALUE_TEXT_ANTIALIAS_LCD_VBGR`, and `VALUE_TEXT_ANTIALIAS_GASP` value constants (which are described in Robert Eckstein's article). According to Bug 6274842 "RFE: Provide a means for a custom look and feel to use desktop font antialiasing settings," however, it may be a while before Java provides the API that custom look and feels need to automatically detect changes to and use the underlying desktop's settings for text anti-aliasing.

NumberFormat and Rounding Modes

The java.text.NumberFormat and java.txt.DecimalFormat classes are used to format numeric values. As evidenced by Bug 4092330 "RFE: Precision, rounding in NumberFormat," it has long been desired for these classes to support the specification of a rounding mode other than the half-even default.

Java SE 6 satisfies this desire by introducing new public void setRoundingMode (RoundingMode roundingMode) and public RoundingMode getRoundingMode() methods into NumberFormat and DecimalFormat. Each class's setRoundingMode() method throws a NullPointerException if you pass null to roundingMode. NumberFormat's setRoundingMode() and getRoundingMode() methods throw UnsupportedOperationException if you attempt to invoke these methods from another NumberFormat subclass that does not override them (java.text.ChoiceFormat, for example). The application shown in Listing 1-3 demonstrates these methods.

Listing 1-3. *NumberFormatRounding.java*

```java
// NumberFormatRounding.java

import java.math.*;

import java.text.*;

public class NumberFormatRounding
{
   public static void main (String [] args)
   {
      NumberFormat nf = NumberFormat.getNumberInstance ();
      nf.setMaximumFractionDigits (2);

      System.out.println ("Default rounding mode: "+nf.getRoundingMode ());
      System.out.println ("123.454 rounds to "+nf.format (123.454));
      System.out.println ("123.455 rounds to "+nf.format (123.455));
      System.out.println ("123.456 rounds to "+nf.format (123.456));
      System.out.println ();

      nf.setRoundingMode (RoundingMode.HALF_DOWN);
      System.out.println ("Rounding mode: "+nf.getRoundingMode ());
      System.out.println ("123.454 rounds to "+nf.format (123.454));
      System.out.println ("123.455 rounds to "+nf.format (123.455));
      System.out.println ("123.456 rounds to "+nf.format (123.456));
      System.out.println ();
```

```
        nf.setRoundingMode (RoundingMode.FLOOR);
        System.out.println ("Rounding mode: "+nf.getRoundingMode ());
        System.out.println ("123.454 rounds to "+nf.format (123.454));
        System.out.println ("123.455 rounds to "+nf.format (123.455));
        System.out.println ("123.456 rounds to "+nf.format (123.456));
        System.out.println ();

        nf.setRoundingMode (RoundingMode.CEILING);
        System.out.println ("Rounding mode: "+nf.getRoundingMode ());
        System.out.println ("123.454 rounds to "+nf.format (123.454));
        System.out.println ("123.455 rounds to "+nf.format (123.455));
        System.out.println ("123.456 rounds to "+nf.format (123.456));
    }
}
```

The source code uses three values: 123.454, 123.455, and 123.456. The first example uses the default half-even rounding mode, which rounds toward the nearest neighbor, or rounds toward the even neighbor if both neighbors are equidistant. Assuming that these values represent 123 dollars and 45.4, 45.5, or 45.6 cents, the default mode is appropriate because it minimizes cumulative errors (statistically) when repeatedly applied to a sequence of calculations. For this reason, half-even is known as *bankers' rounding* (it is used mainly in the United States). However, you might prefer a different rounding mode for your application if the value to be formatted represents something other than currency.

For example, you could work with half-down rounding, which is similar to half-even, except that it rounds down instead of to the even neighbor when both neighbors are equidistant. You could also work with floor and ceiling rounding, to round toward negative and positive infinity, respectively. To see these rounding modes in action, compile NumberFormatRounding.java and run the application. You will see the following output:

```
Default rounding mode: HALF_EVEN
123.454 rounds to 123.45
123.455 rounds to 123.46
123.456 rounds to 123.46

Rounding mode: HALF_DOWN
123.454 rounds to 123.45
123.455 rounds to 123.45
123.456 rounds to 123.46
```

```
Rounding mode: FLOOR
123.454 rounds to 123.45
123.455 rounds to 123.45
123.456 rounds to 123.45

Rounding mode: CEILING
123.454 rounds to 123.46
123.455 rounds to 123.46
123.456 rounds to 123.46
```

> ■**Note** In addition to the `NumberFormat` and `DecimalFormat` enhancements, Java SE 6 adds new `public static Locale[] getAvailableLocales()`, `public static final DecimalFormatSymbols getInstance()`, `public static final DecimalFormatSymbols getInstance(Locale locale)`, `public String getExponentSeparator()`, and `public void setExponentSeparator(String exp)` methods to the `java.text.DecimalFormatSymbols` class.

Improved File Infrastructure

Java SE 6 extends the `java.io.File` class with several new methods, which Chapter 2 explores. It also improves `File`'s infrastructure on Microsoft Windows platforms.

One improvement is that Windows devices (such as NUL, AUX, and CON) are no longer considered to be files, which results in `File`'s `public boolean isFile()` method returning false when confronted with a device name. For example, `System.out.println (new File ("CON").isFile ());` outputs true under Java 5 and false under Java SE 6.

Another improvement involves the critical message dialog box. Prior to Java SE 6, a `File` method's attempt to access a drive whose removable media (a CD or a floppy disk, for example) was absent resulted in Windows presenting a critical message dialog box, which provided the option to retry the operation, after the media was presumably inserted into the drive. If you were remotely monitoring this program, you obviously had a problem when confronted by the dialog box: you were not present to insert the disk and click the dialog box's Continue button. For this reason, Java SE 6 prevents this dialog box from appearing, and fails the operation by having the method return a suitable value. For example, if you try to execute `System.out.println (new File ("A:\\someFile.txt").exists ());` without a floppy disk in the A: drive, a dialog box will not appear, and `exists()` will return false.

■**Note** `File`'s infrastructure now supports long pathnames on Windows platforms, where each pathname element is Windows-limited to 260 characters. Check out Bug 4403166 "File does not support long paths on Windows NT."

Continuing with improvements to `File`'s infrastructure, Bug 6198547 "File.create-NewFile() on an existing directory incorrectly throws IOException (win)" points out that invoking `File`'s `public boolean createNewFile()` method with the name of the file to be created matching the name of an existing directory results in a thrown `java.io.IOException`, instead of false being returned (as stated in the JDK documentation). Java SE 6 corrects this discrepancy by having this method return false.

■**Note** Java SE 6 also improves Mac Mac OS X's `File` infrastructure by addressing Bug 6395581 "File.listFiles() is unable to read nfs-mounted directory (Mac OS X)." `File`'s `listFiles()` methods now reads Mac OS X's NFS-mounted directories.

As a final Windows-specific improvement, `File`'s `public long length()` method no longer returns 0 for special files, such as `pagefile.sys`. Bug 6348207 "File.length() reports a length of 0 for special files hiberfil.sys and pagefile.sys (win)" documents this improvement. Although you might not find this improvement helpful, you will probably benefit from the platform-independent improvement offered by Bug 4809375 "File.deleteOnExit() should be implemented with shutdown hooks."

■**Note** In addition to introducing new methods and making the aforementioned improvements to `File`, Java SE 6 has also deprecated this class's `public URL toURL()` method. The JDK documentation offers this explanation: "This method does not automatically escape characters that are illegal in URLs. It is recommended that new code convert an abstract pathname into a URL by first converting it into a URI via the `toURI` method, and then converting the URI into a URL via the `URI.toURL` method."

Window Icon Images

The `java.awt.Frame` class has always had a `public void setIconImage(Image image)` method to specify a frame window's icon image, which appears on the left side of the frame window's title bar, and a companion `public Image getIconImage()` method to return this image. Although these methods are also available to `Frame`'s `javax.swing.JFrame` subclass, which overrides `setIconImage()` to invoke its superclass version and then fire a property change event in pre-Java SE 6, they are not available to

`javax.swing.JDialog`. An application whose frame window displays a custom icon, but whose dialogs do not, gives the impression that the dialogs do not belong to the application.

Bug 4913618 "Dialog doesn't inherit icon from its parent frame" documents this problem. Java SE 6 provides a solution that results in a dialog window now inheriting the icon from its parent frame window. Because this might be problematic if you want to supply a different icon for some specific dialog, Java SE 6 also adds a new `public void setIconImage(Image image)` method to the `java.awt.Window` class. This method allows you to specify a custom icon for a dialog window.

Modern operating systems typically display an application's icon in multiple places. In addition to a window's title bar, an icon can appear on the taskbar, on a task switcher (such as the Windows XP task switcher), beside a task name in a list of running tasks (such as the Applications tab of the Windows XP Windows Task Manager), and so on. In some places, the icon will appear at a different size. For example, the icon on the Windows XP task switcher is larger than the icon on the window's title bar. Prior to Java SE 6, the icon image assigned to the frame window via `setIconImage()` was scaled to appear larger on the taskbar; the result often looked terrible. This problem is documented by Bug 4721400 "Allow to specify 32x32 icon for JFrame (or Window)."

Java SE 6 provides a solution by adding new `public void setIconImages(List<? extends Image> icons)` and `public List<Image> getIconImages()` methods to `Window`. The former method lets you specify a list of icon images for display on the window's title bar and in other contexts, such as the taskbar or a task switcher. Prior to an icon being selected for a specific context, the `icons` list is scanned from the beginning for the first icon that has appropriate dimensions.

To demonstrate solutions to the dialog-does-not-inherit-icon and one-icon-for-all-contexts problems, I've created an application that creates a small solid icon and a big striped icon, assigns them to a frame window, and displays the frame window and a dialog. Listing 1-4 presents the source code.

Listing 1-4. *WindowIcons.java*

```java
// WindowIcons.java

import java.awt.*;
import java.awt.image.*;

import java.util.*;

import javax.swing.*;

public class WindowIcons extends JFrame
{
```

```java
final static int BIG_ICON_WIDTH = 32;
final static int BIG_ICON_HEIGHT = 32;
final static int BIG_ICON_RENDER_WIDTH = 20;

final static int SMALL_ICON_WIDTH = 16;
final static int SMALL_ICON_HEIGHT = 16;
final static int SMALL_ICON_RENDER_WIDTH = 10;

public WindowIcons ()
{
   super ("Window Icons");
   setDefaultCloseOperation (EXIT_ON_CLOSE);

   ArrayList<BufferedImage> images = new ArrayList<BufferedImage> ();

   BufferedImage bi;
   bi = new BufferedImage (SMALL_ICON_WIDTH, SMALL_ICON_HEIGHT,
                           BufferedImage.TYPE_INT_ARGB);
   Graphics g = bi.getGraphics ();
   g.setColor (Color.black);
   g.fillRect (0, 0, SMALL_ICON_RENDER_WIDTH, SMALL_ICON_HEIGHT);
   g.dispose ();
   images.add (bi);

   bi = new BufferedImage (BIG_ICON_WIDTH, BIG_ICON_HEIGHT,
                           BufferedImage.TYPE_INT_ARGB);
   g = bi.getGraphics ();
   for (int i = 0; i < BIG_ICON_HEIGHT; i++)
   {
       g.setColor (((i & 1) == 0) ? Color.black : Color.white);
       g.fillRect (0, i, BIG_ICON_RENDER_WIDTH, 1);
   }
   g.dispose ();
   images.add (bi);

   setIconImages (images);

   setSize (250, 100);
   setVisible (true);

   // Create and display a modeless Swing dialog via an anonymous inner
   // class.

   new JDialog (this, "Arbitrary Dialog")
```

```
         {
           {
              setSize (200, 100);
              setVisible (true);
           }
         };
      }

   public static void main (String [] args)
   {
      Runnable r = new Runnable ()
                 {
                     public void run ()
                     {
                        new WindowIcons ();
                     }
                 };
      EventQueue.invokeLater (r);
   }
}
```

In response to Bug 6339074 "Improve icon support," which states that icons should support transparency, the application renders only part of each icon in an ARGB buffer, to see if transparency is honored under Windows XP Service Pack (SP) 2. According to Figure 1-2, transparency is honored.

Figure 1-2. *The small icon appears on the title bar of the frame and dialog windows, and on the taskbar, but not on the task switcher. The big icon appears on the task switcher.*

Window Minimum Size

Bug 4320050 "Minimum size for java.awt.Frame is not being enforced" describes a long-standing GUI problem where it is not possible to establish a window's minimum size. If the minimum size could be set, you could then prevent your application's users from resizing the main window below the minimum size (and avoid phone calls from inexperienced and panicked users who can no longer access the GUI).

Java SE 6 adds a new `public void setMinimumSize(Dimension minimumSize)` method to `Window`, to let you enforce a minimum size. A subsequent call to `Window`'s inherited `public Dimension getMinimumSize()` method returns the new minimum size. If the window's size prior to this call is smaller than the minimum size, the window is automatically enlarged to honor the minimum. The following code fragment sets a frame window's minimum size to 400-by-300 pixels:

```
Frame frame = new Frame ("Some window title");

// Do not allow the user to resize the frame below 400
// pixels horizontally and 300 pixels vertically.

frame.setMinimumSize (new Dimension (400, 300));
```

■**Note** `Window` overrides the `Component` class's `public void setSize(Dimension d)`, `public void setSize(int width, int height)`, `public void setBounds(int x, int y, int width, int height)`, and `public void setBounds(Rectangle r)` methods to prevent a window from being sized below its minimum size. If a method is called with a width or height that is less than the current minimum size, the method enlarges the width or height.

Interruptible I/O Switch for Solaris

Solaris native-thread implementations of the virtual machine take advantage of the Solaris operating system's support for interruptible I/O. As a result, a thread that is blocked on an I/O operation can be interrupted via a call to the `Thread` class's `public void interrupt()` method on the blocked thread's `Thread` object; a `java.io.InterruptedIOException` is thrown from the interrupted thread.

Bug 4154947 "JDK 1.1.6, 1.2/Windows NT: Interrupting a thread blocked does not unblock IO" explains the difficulty in trying to implement interruptible I/O on the Windows platform. Because it could prove impossible to provide this feature on Windows, and because having interruptible I/O support available to Solaris virtual machines but not available to Windows virtual machines violates Java's cross-platform nature, Java SE 6 introduces a new `UseVMInterruptibleIO` HotSpot option switch to turn off

interruptible I/O on the Solaris virtual machine. Interruptible I/O is still enabled by default (it might be disabled by default in Java SE 7). You can explicitly disable interruptible I/O by specifying `-XX:-UseVMInterruptibleIO` when starting the Solaris virtual machine. For more information, check out Bug 4385444 "(spec) InterruptedIOException should not be required by platform specification (sol)."

ZIP and JAR Files

Java SE 6 introduces various enhancements in the context of ZIP and JAR files. From the API perspective, the `java.util.zip` package has new `DeflaterInputStream` and `InflaterOutputStream` classes. These classes allow an application to send compressed data over a network. Data is compressed into packets via `DeflaterInputStream`, and the packets are sent over the network to a destination, where they are then decompressed via `InflaterOutputStream`.

Non-API enhancements include allowing ZIP files to contain more than 64,000 entries on all platforms. For Windows platforms, the upper limit of 2,036 concurrently open ZIP files has been removed, and the limit is now determined by the platform; see Bug 6423026 "Java.util.zip doesn't allow more than 2036 zip files to be concurrently open on Windows." Also, filenames longer than 256 characters are supported; see Bug 6374379 "ZipFile class cannot open zip files with long filenames."

Regarding JAR files, the `jar` tool has been enhanced so that the timestamps of extracted files match the timestamps that appear in the archive. Prior to Java SE 6, an extracted file's timestamp was set to the current time. Check out Appendix B to see what else has changed for the `jar` tool.

Ownerless Windows

Chapter 3 introduces Java SE 6's new modality model and API. To work properly, this model depends on *ownerless windows,* which are windows without parent windows; a frame window created by the `public JFrame()` constructor is an example of an owner-less window.

It turns out that early attempts to support ownerless windows were problematic. For example, Bug 4256840 "Exception when using the no-argument Window() constructor on win32," and Bug 4262946 "API Change: remove constructors for ownerless Windows in java.awt.Window" revealed that the introduction of ownerless windows into Java 1.3 (Kestrel) via `public Window()` and `public Window(GraphicsConfiguration gc)` constructors led to ownerless `Windows` not showing up in the array returned by `Frame`'s `public static Frame[] getFrames()` method. Suddenly, an automation tool could not access an application's entire tree of GUI components.

To address this problem, the `JDialog` class includes constructors such as `public JDialog(Frame owner)`. If you pass `null` to `owner`, a shared hidden frame window is chosen

as the owner of the dialog. Automation tools can get access to this frame window. Unfortunately, as pointed out in Bug 6300062 "JDialog need to support true parent-less mode," this frame window causes problems for the new modality model. You might want to read Chapter 3's modality model/API introduction before reading this bug report to first grasp the basics.

Java SE 6 solves both the modality problem and the automation tool problem as follows:

- By allowing you to pass `null` to the `owner` parameter in any of `Window`'s constructors, so that these windows can be ownerless

- By allowing you to pass `null` to the `owner` parameter in any of `java.awt.Dialog`'s constructors, so that these dialog windows can be ownerless

- By introducing several new `JDialog` constructors—the first parameter is of type `Window` (`public JDialog(Window owner)`, for example)—that let you pass `null` to `owner` for true ownerless Swing dialog windows

- By introducing two new methods into the `Window` class: `public static Window[] getWindows()`, which lets an automation tool obtain an array of all ownerless and owned windows, and `public static Window[] getOwnerlessWindows()`, which lets this tool obtain an array of ownerless windows only

Listing 1-5 presents an application that demonstrates the `public static Window[] getWindows()` and `public static Window[] getOwnerlessWindows()` methods.

Listing 1-5. *Windows.java*

```
// Windows.java

import java.awt.*;

import javax.swing.*;

public class Windows
{
   public static void main (String [] args)
   {
      // Create a pseudo-ownerless Swing dialog (its owner is a hidden shared
      // frame window).

      JDialog d1 = new JDialog ((JFrame) null, "Dialog 1");
      d1.setName ("Dialog 1");
```

```
      // Create a true ownerless Swing dialog.

      JDialog d2 = new JDialog ((Window) null, "Dialog 2");
      d2.setName ("Dialog 2");

      // Create an ownerless frame.

      Frame f = new Frame ();
      f.setName ("Frame 1");

      // Create a window owned by the frame.

      Window w1 = new Window (f);
      w1.setName ("Window 1");

      // Create an ownerless window.

      Window w2 = new Window (null);
      w2.setName ("Window 2");

      // Output lists of all windows, ownerless windows, and frame windows.

      System.out.println ("ALL WINDOWS");
      Window [] windows = Window.getWindows ();
      for (Window window: windows)
          System.out.println (window.getName ()+": "+window.getClass ());
      System.out.println ();

      System.out.println ("OWNERLESS WINDOWS");
      Window [] ownerlessWindows = Window.getOwnerlessWindows ();
      for (Window window: ownerlessWindows)
          System.out.println (window.getName ()+": "+window.getClass ());
      System.out.println ();

      System.out.println ("FRAME WINDOWS");
      Frame [] frames = Frame.getFrames ();
      for (Frame frame: frames)
          System.out.println (frame.getName ()+": "+frame.getClass ());
   }
}
```

After compiling the source code and running this application, you'll discover the following output, which reveals that Dialog 1 is not a true ownerless window:

```
ALL WINDOWS
frame0: class javax.swing.SwingUtilities$SharedOwnerFrame
Dialog 1: class javax.swing.JDialog
Dialog 2: class javax.swing.JDialog
Frame 1: class java.awt.Frame
Window 1: class java.awt.Window
Window 2: class java.awt.Window

OWNERLESS WINDOWS
frame0: class javax.swing.SwingUtilities$SharedOwnerFrame
Dialog 2: class javax.swing.JDialog
Frame 1: class java.awt.Frame
Window 2: class java.awt.Window

FRAME WINDOWS
frame0: class javax.swing.SwingUtilities$SharedOwnerFrame
Frame 1: class java.awt.Frame
```

Navigable Sets

Chapter 2 introduces Java SE 6's enhanced collections framework. One enhancement worth mentioning here is a new `java.util.NavigableSet<E>` interface, which extends the older `java.util.SortedSet<E>` interface and facilitates navigating through an ordered set-based collection.

A navigable set can be accessed and traversed in ascending order via the `Iterator<E>` `iterator()` method, and in descending order via the `Iterator<E> descendingIterator()` method. It can return the closest matches for given search targets via methods `public E` `ceiling(E e)`, `public E floor(E e)`, `public E higher(E e)`, and `public E lower(E e)`. By default, these closest-match methods find the closest match in ascending order. To find a closest match in descending order, first obtain a reverse-order view of the set via the `NavigableSet<E> descendingSet()` method. Listing 1-6 presents an application that demonstrates `descendingSet()` and the four closest-match methods, with comments that describe each closest-match method in detail.

Listing 1-6. *CityNavigator.java*

```java
// CityNavigator.java

import java.util.*;

public class CityNavigator
{
   static NavigableSet<String> citiesSet;

   public static void main (String [] args)
   {
      String [] cities =
      {
         "Beijing",
         "Berlin",
         "Baghdad",
         "Buenos Aires",
         "Bangkok",
         "Belgrade"
      };

      // Create and populate a navigable set of cities.

      citiesSet = new TreeSet<String> ();
      for (String city: cities)
           citiesSet.add (city);

      // Dump the city names in ascending order. Behind the scenes, the
      // following code is implemented in terms of
      //
      // Iterator iter = citiesSet.iterator ();
      // while (iter.hasNext ())
      //    System.out.println (iter.next ());

      System.out.println ("CITIES IN ASCENDING ORDER");
      for (String city: citiesSet)
           System.out.println ("  "+city);
      System.out.println ();

      // Dump the city names in descending order. Behind the scenes, the
      // following code is implemented in terms of
```

```
      //
      // Iterator iter = citiesSet.descendingSet.iterator ();
      // while (iter.hasNext ())
      //     System.out.println (iter.next ());

      System.out.println ("CITIES IN DESCENDING ORDER");
      for (String city: citiesSet.descendingSet ())
          System.out.println ("  "+city);
      System.out.println ();

      // Demonstrate the closest-match methods in ascending order set.

      System.out.println ("CLOSEST-MATCH METHODS/ASCENDING ORDER DEMO");
      outputMatches ("Berlin");
      System.out.println ();

      outputMatches ("C");
      System.out.println ();

      outputMatches ("A");
      System.out.println ();

      // Demonstrate closest-match methods in descending order set.

      citiesSet = citiesSet.descendingSet ();
      System.out.println ("CLOSEST-MATCH METHODS/DESCENDING ORDER DEMO");
      outputMatches ("Berlin");
      System.out.println ();

      outputMatches ("C");
      System.out.println ();

      outputMatches ("A");
      System.out.println ();
   }

   static void outputMatches (String city)
   {
      // ceiling() returns the least element in the set greater than or equal
      // to the given element (or null if the element does not exist).

      System.out.println ("  ceiling('"+city+"'): "+citiesSet.ceiling (city));
```

```
        // floor() returns the greatest element in the set less than or equal to
        // the given element (or null if the element does not exist).

        System.out.println ("  floor('"+city+"'): "+citiesSet.floor (city));

        // higher() returns the least element in the set strictly greater than
        // the given element (or null if the element does not exist).

        System.out.println ("  higher('"+city+"'): "+citiesSet.higher (city));

        // lower() returns the greatest element in the set strictly less than
        // the given element (or null if the element does not exist).

        System.out.println ("  lower('"+city+"'): "+citiesSet.lower (city));
    }
}
```

As shown in the source code, the closest-match methods return set elements that satisfy various conditions. For example, lower() returns the element that is greater than all other set elements, except for the element described by lower()'s argument; the method returns null if there is no such element. Although this description is intuitive when you consider a set that is ordered in ascending order, intuition fails somewhat when you consider the set ordered in descending order. For example, in the following output, Belgrade is lower than Berlin in ascending order, and Buenos Aires is lower than Berlin in descending order:

```
CITIES IN ASCENDING ORDER
  Baghdad
  Bangkok
  Beijing
  Belgrade
  Berlin
  Buenos Aires
```

```
CITIES IN DESCENDING ORDER
  Buenos Aires
  Berlin
  Belgrade
  Beijing
  Bangkok
  Baghdad

CLOSEST-MATCH METHODS/ASCENDING ORDER DEMO
  ceiling('Berlin'): Berlin
  floor('Berlin'): Berlin
  higher('Berlin'): Buenos Aires
  lower('Berlin'): Belgrade

  ceiling('C'): null
  floor('C'): Buenos Aires
  higher('C'): null
  lower('C'): Buenos Aires

  ceiling('A'): Baghdad
  floor('A'): null
  higher('A'): Baghdad
  lower('A'): null

CLOSEST-MATCH METHODS/DESCENDING ORDER DEMO
  ceiling('Berlin'): Berlin
  floor('Berlin'): Berlin
  higher('Berlin'): Belgrade
  lower('Berlin'): Buenos Aires

  ceiling('C'): Buenos Aires
  floor('C'): null
  higher('C'): Buenos Aires
  lower('C'): null

  ceiling('A'): null
  floor('A'): Baghdad
  higher('A'): null
  lower('A'): Baghdad
```

■**Note** Here are a few other interesting changes in Java SE 6:

- Java SE 6 changes the class file version number to 50.0 because it supports split verification (see Appendix B).

- Java SE 6's `jarsigner`, `keytool`, and `kinit` security tools no longer echo passwords to the screen.

- The `javax.swing.text.Segment` class, which allows fast access to a segment of text, now implements the `CharSequence` interface. You can use `Segment` in regular-expression contexts, for example.

Java SE 6, Update 1 and Update 2

Following the initial release of Java SE 6 (which is the focus of this book), Sun released its first Java SE 6 update to introduce a number of bug fixes. This update release specifies 6u01 as its external version number, and 1.6.0_01-b06 (where *b* stands for build) as its internal version number.

One bug that has been fixed in 6u01 concerns memory leak problems with several methods. For example, the `Thread` class specifies a `public static Map<Thread, StackTraceElement[]> getAllStackTraces()` method that returns a map of stack traces for all live threads. Also, the `java.lang.management.ThreadMXBean` interface specifies several `getThreadInfo()` methods that return thread information. According to Bug 6434648 "Native memory leak when use Thread.getAllStackTraces()," all of these methods have a memory leak that leads to an `OutOfMemoryError`. You can reproduce this problem, which has been solved in this update release, by running the following application (which might run for a considerable period of time before `OutOfMemoryError` is thrown) on the initial release of Java SE 6:

```java
public class TestMemoryLeak
{
    public static void main(String[] args)
    {
        while (true)
        {
            Thread.getAllStackTraces();
        }

    }
}
```

Another bug that has been fixed in 6u01 is Bug 6481004 "SplashScreen.getSplashScreen() fails in Web Start context." According to this bug, migrating a stand-alone application that uses the Splash Screen API to Java Web Start results in a `java.security.AccessControlException` being thrown. This exception is thrown as a result of the `System.loadLibrary("splashscreen")` method call in the `public static synchronized SplashScreen getSplashScreen()` method not being placed inside a `doPrivileged()` block.

The Java SE 6 Update Release Notes page (`http://java.sun.com/javase/6/webnotes/ReleaseNotes.html`) provides a complete list of all the bugs that have disappeared in the 6u01 update.

While this chapter was being written, a second Java SE 6 update was released. Although this update was rumored to contain a slimmed-down JRE, as pointed out by the posting on TheServerSide.com titled "Rumor: Java 6 update 2 will be 2-4MB?" (`http://www.theserverside.com/news/thread.tss?thread_id=45377`), the second update offered nothing quite so dramatic. This rumor was most likely based on the much-discussed Consumer JRE, which Chet Haase discusses in his "Consumer JRE: Leaner, Meaner Java Technology" article (`http://java.sun.com/developer/technicalArticles/javase/consumerjre/`).

To see what the second update has to offer, check out Sun's Java SE 6 Update Release Notes page.

Summary

Java SE 6 (formerly known as Mustang) officially arrived on December 11, 2006. This release contains many new and improved features that will benefit Java developers for years to come.

Java SE 6 was developed under JSR 270, which presents various themes. These themes include compatibility and stability; diagnosability, monitoring, and management; ease of development; enterprise desktop; XML and web services; and transparency.

JSR 270 identifies various component JSRs. These JSRs include JSR 105 XML Digital Signature APIs, JSR 199 Java Compiler API, JSR 202 Java Class File Specification Update, JSR 221 JDBC 4.0 API Specification, JSR 222 Java Architecture for XML Binding (JAXB) 2.0, JSR 223 Scripting for the Java Platform, JSR 224 Java API for XML-Based Web Services (JAX-WS) 2.0, JSR 268 Java Smart Card I/O API, and JSR 269 Pluggable Annotation Processing API. Although not identified by JSR 270, JSR 173 Streaming API for XML, JSR 181 Web Services Metadata for the Java Platform, and JSR 250 Common Annotations for the Java Platform are also component JSRs.

Java SE 6 provides many features that set it apart from its predecessors. Some of these features were explored in this chapter, and include a trio of new action keys and a method to hide/show action text, the ability to clear a button group's selection, reflection

enhancements, the GroupLayout layout manager, an Image I/O GIF writer plug-in, incremental improvements to the String class, LCD text support, new NumberFormat methods for working with rounding modes, an improved File class infrastructure, window icon images, the ability to specify a minimum window size, an interruptible I/O switch for Solaris, DeflatorInputStream and InflatorOutputStream classes added to the java.util.zip package, ownerless windows, and navigable sets.

Following the initial release of Java SE 6 (which is the focus of this book), Sun released a pair of updates that primarily fix bugs.

Test Your Understanding

How well do you understand Java SE 6 thus far? Test your understanding by answering the following questions and performing the following exercises. (The answers are presented in Appendix D.)

1. Why does Sun refer to Java SE 6 instead of J2SE 6.0?

2. Identify the themes of Java SE 6.

3. Does Java SE 6 include internationalized resource identifiers (IRIs)?

4. What is the purpose of Action's new DISPLAYED_MNEMONIC_INDEX_KEY constant?

5. Why should you create a Swing program's GUI only on the event-dispatching thread?

6. How do you establish a window's minimum size?

7. Describe each of NavigableSet<E>'s closest-match methods.

8. Does public JDialog(Frame owner) create a true ownerless window when owner is null?